

3

INTERNET PRINCIPLES

IF YOU ARE reading this book, we assume that you probably use the Internet regularly—for browsing the web, reading and sending email, listening to music. But familiarity with the Internet is very much on a gradient. Using the Internet daily is a first step in understanding it; developing software or Things that speak to the Internet is another; and developing or debugging the software that runs the Internet itself is yet another.

This gradient of familiarity also applies to the authors of this book. Perhaps, like Adrian, you have implemented a full TCP/IP stack, several times. Perhaps, like Hakim, you're not entirely sure what that jargon means.

Whether you are a developer, engineer, artist, or entrepreneur, having a high-level view of the different components and technologies that make up what we call the Internet will help you understand the possibilities and the current limitations of what you can do with the Internet of Things.

This chapter is a short, high-level survey, designed to be readable rather than complete. Let's start by looking at an example of how communication over long distances could work in the real world and then compare it to how information is transmitted across the virtual world of the Internet.

INTERNET COMMUNICATIONS: AN OVERVIEW

Suppose that you wanted to send a message to the authors of this book, but you didn't have the postal address, and you didn't have any way to look up our phone number (because in this example you don't have the Internet).

You remember that we're from the UK, and London is the biggest city in the UK. So you send a postcard to your cousin Bob, who lives there.

Your cousin sees that the postcard is for some crazy hardware and technology people. So he puts the postcard in an envelope and drops it off at the London Hackspace because the guys there probably know what to do with it.

At the Hackspace, Jonty picks up the envelope and sees that it's for some people in Liverpool. Like all good Londoners, Jonty never goes anywhere to the north of Watford, but he remembers that Manchester is in the north too. So he calls up the Manchester Digital Laboratory (MadLab), opens the envelope to read the contents, and says, "Hey, I've got this message for Adrian and Hakim in Liverpool. Can you pass it on?"

The guys at MadLab ask whether anyone knows who we are, and it turns out that Hwa Young does. So the next time she comes to Liverpool, she delivers the postcard to us.

IP

The preceding scenario describes how the Internet Protocol (IP) works. Data is sent from one machine to another in a packet, with a destination address and a source address in a standardised format (a "protocol"). Just like the original sender of the message in the example, the sending machine doesn't always know the best route to the destination in advance. Most of the time, the packets of data have to go through a number of intermediary machines, called *routers*, to reach their destination. The underlying networks aren't always the same: just as we used the phone, the postal service, and delivery by hand, so data packets can be sent over wired or wireless networks, through the phone system, or over satellite links.

In our example, a postcard was placed in an envelope before getting passed onwards. This happens with Internet packets, too. So, an *IP packet* is a block of data along with the same kind of information you would write on a

physical envelope: the name and address of the server, and so on. But if an IP packet ever gets transmitted across your local wired network via an Ethernet cable—the cable that connects your home broadband router or your office local area network (LAN) to a desktop PC—then the whole packet will get bundled up into another type of envelope, an *Ethernet Frame*, which adds additional information about how to complete the last few steps of its journey to your computer.

Of course, it's possible that your cousin Bob didn't know about the London Hackspace, and then maybe the message would have got stuck with him. You would have had no way to know whether it got there. This is how IP works. There is no guarantee, and you can send only what will fit in a single packet.

TCP

What if you wanted to send longer messages than fit on a postcard? Or wanted to make sure your messages got through?

What if everyone agreed that postcards written in green ink meant that we cared about whether they arrived. And that we would always number them, so if we wanted to send longer messages, we could. The person at the other end would be able to put the messages in order, even if they got delivered in the wrong order (maybe you were writing your letter over a number of days, and the day you passed the fifth one on to cousin Bob, he happened to visit Liverpool and passed on that postcard without relaying through London Hackspace or MadLab). We would send back postcard notifications that just told you which postcards we had received, so you could resend any that went missing.

That is basically how the Transmission Control Protocol (TCP) works. The simplest transport protocol on the Internet, TCP is built on top of the basic IP protocol and adds sequence numbers, acknowledgements, and retransmissions. This means that a message sent with TCP can be arbitrarily long and give the sender some assurance that it actually arrived at the destination intact.

Because the combination of TCP and IP is so useful, many services are built on it in turn, such as email and the HTTP protocol that transmits information across the World Wide Web.

THE IP PROTOCOL SUITE (TCP/IP)

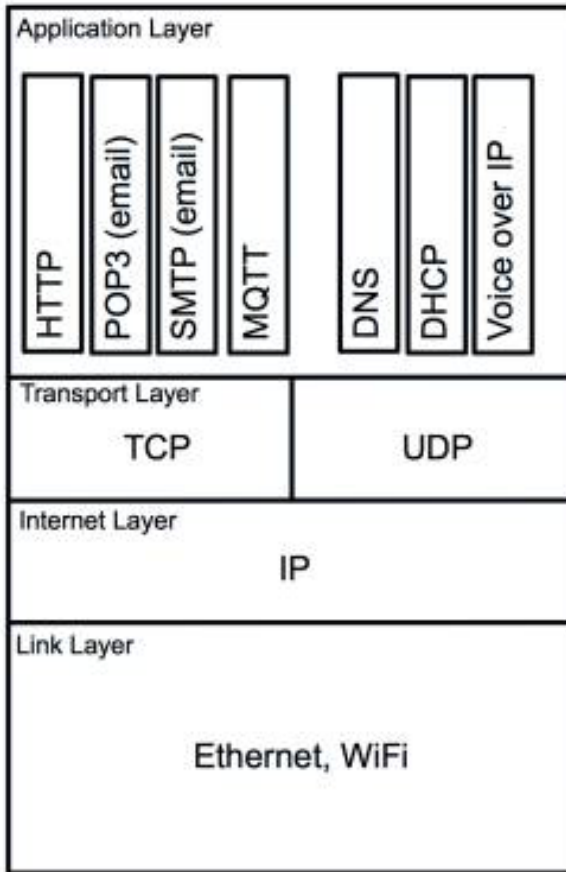
The combination of TCP and IP is so ubiquitous that we often refer simply to “TCP/IP” to describe a whole suite or stack of protocols layered on top of each other, each layer building on the capabilities of the one below.

- The low-level protocols at the *link layer* manage the transfer of bits of information across a network link. This could be by an Ethernet cable, by WiFi, or across a telephone network, or even by short-range radio standards such as IEEE 802.15.4 designed to carry data over the Personal Area Network (PAN), that is to say between devices carried by an individual.
- The *Internet layer* then sits on top of these various links and abstracts away the gory details in favour of a simple destination address.
- Then TCP, which lives in the *transport layer*, sits on top of IP and extends it with more sophisticated control of the messages passed.
- Finally, the *application layer* contains the protocols that deal with fetching web pages, sending emails, and Internet telephony. Of these, HTTP is the most ubiquitous for the web, and indeed for communication between Internet of Things devices. We look at standards such as MQTT briefly in Chapter 7.

UDP

As you can see, TCP is not the only protocol in the transport layer. Unlike TCP, but as with IP itself, in UDP each message may or may not arrive. No handshake or retransmission occurs, nor is there any delay to wait for messages in sequence. These limitations make TCP preferable for many of the tasks that Internet of Things devices will be used for.

The lack of overhead, however, makes UDP useful for applications such as streaming data, which can cope with minor errors but doesn't like delays. Voice over IP (VoIP)—computer-based telephony, such as Skype—is an example of this: missing one packet might cause a tiny glitch in the sound quality, but waiting for several packets to arrive in the right order could make the speech too jittery to be easy to understand. UDP is also the transport for some very important protocols which provide common, low-level functionality, such as DNS and DHCP, which relate to the discovery and resolution of devices on the network. We look at this topic in detail in the next section.



The Internet Protocol suite.

IP ADDRESSES

We mentioned earlier that the Internet Protocol knows the addresses of the destination and source devices. But what does an “address” consist of? Here is a typical human (or in this case, hobbit) address:

Bilbo Baggins
“Bag End”, Bagshot Row
Hobbiton
The Shire
Middle Earth

In the world of low-level computer networking, however, numbers are much easier to deal with. So, IP addresses are numbers. In Internet Protocol version 4 (IPv4), almost 4.3 billion IP addresses are possible—4,294,967,296 to be precise, or 2^{32} . Though that is convenient for computers, it's tough for humans to read, so IP addresses are usually written as four 8-bit numbers separated by dots (from 0.0.0.0 to 255.255.255.255)—for example, 192.168.0.1 (which is often the address of your home router) or 8.8.8.8 (which is the address of one of Google's DNS servers).

This “dotted quad” is still exactly equivalent to the 32-bit number. As well as being simply easier for humans to remember, it is also easier to infer information about the address by grouping certain blocks of addresses together. For example,

8.8.8.x — One of several IP ranges assigned to Google.

192.168.x.x — A range assigned for private networks. Your home or office network router may well assign IP addresses in this range.

10.x.x.x — Another private range.

Every machine on the Internet has at least one IP address. That means every computer, every network-connected printer, every smartphone, and every Internet of Things device has one. If you already have a Raspberry Pi, an Arduino board, or any of the other microcontrollers described in Chapters 3 and 4, they will expect to get their own IP address, too. When you consider this fact, those 4 billion addresses suddenly look as if they might not be enough.

The private ranges such as 192.168.x.x offer one mitigation to this problem. Your home or office network might have only one publicly visible IP address. However, you could have all the IP addresses in the range 192.168.0.0 to 192.168.255.255 ($2^{16} = 65,536$ addresses) assigned to distinct devices.

A better solution to this problem is the next generation of Internet Protocol, IPv6, which we look at later in this chapter.

DNS

Although computers can easily handle 32-bit numbers, even formatted as dotted quads they are easy for most humans to forget. The Domain Name System (DNS) helps our feeble brains navigate the Internet. Domain names,

such as the following, are familiar to us from the web, or perhaps from email or other services:

```
google.com
bbc.co.uk
wiley.com
arduino.cc
```

Each domain name has a top-level domain (TLD), like `.com` or `.uk`, which further subdivides into `.co.uk` and `.gov.uk`, and so on. This top-level domain knows where to find more information about the domains within it; for example, `.com` knows where to find `google.com` and `wiley.com`.

The domains then have information about where to direct calls to individual machines or services. For example, the DNS records for `.google.com` know where to point you for the following:

```
www.google.com
mail.google.com
calendar.google.com
```

The preceding examples are all instantly recognizable as website names, which is to say you could enter them into your web browser as, for example, `http://www.google.com`.

But DNS can also point to other services on the Internet—for example:

```
pop3.google.com — For receiving email from Gmail
smtp.google.com — For sending email to Gmail
ns1.google.com — The address of one of Google's many DNS
servers
```

Configuring DNS is a matter of changing just a few settings. Your registrar (the company that sells you your domain name) often has a control panel to change these settings. You might also run your own authoritative DNS server. The settings might contain entries like this one for `roomofthings.com`:

```
book A 80.68.93.60 3h
```

This entry means that the address `book.roomofthings.com` (which hosts the blog for this book) is served by that IP address and will be for the next three hours.

STATIC IP ADDRESS ASSIGNMENT

How do you get assigned an IP address? If you have bought a server-hosting package from an Internet service provider (ISP), you might typically be given a single IP address. But the company itself has been given a block of addresses to assign. Historically, these were ranges of different sizes, typically separated into “classes” of 8 bits, 16 bits, or 24 bits:

Class A — From `0.x.x.x`

Class B — From `128.0.x.x`

Class C — From `192.0.0.x`

The class C ranges had a mere 8 bits (256 addresses) assigned to them, while the class A ranges had many more addresses and would therefore be given only to the very largest of Internet organisations. The rigid separation of address ranges into classes was not very efficient; every entity would want to keep enough spare addresses for future expansion, but this means that many addresses would remain unused. With the explosion of the number of devices connecting to the Internet (a theme throughout this chapter), the scheme has been superseded since 1993 by Classless Inter-Domain Routing (CIDR), which allows you to specify exactly how many bits of the address are fixed. (See RFCs 1518 and 1519, at <http://tools.ietf.org/rfc/>.) So, the class A addresses we mentioned above would be equivalent to `0.0.0.0/8`, while a class C might be `208.215.179.0/24`.

For example, you saw previously that Google had the range

`8.8.8.x` (which is equivalent to `8.8.8.0/24` in CIDR notation)

Google has chosen to give one of its public DNS servers the address

`8.8.8.8`

from this range, largely because this address is easy to remember.

In many cases, however, the system administrator simply assigns server numbers in order. The administrator makes a note of the addresses and updates DNS records and so on to point to these addresses. We call this kind of address *static* because once assigned it won't change again without human intervention.

Now consider your home network: every time you plug a desktop PC to your router, connect your laptop or phone to the wireless, or switch on your network-enabled printer, this device has to get an IP address (often in the range `192.168.0.0/16`). You *could* assign an address sequentially yourself, but the typical person at home isn't a system administrator and may not keep thorough records. If your brother, who used to use the address `192.168.0.5` but hasn't been home for ages, comes back to find that your new laser printer now has that address, he won't be able to connect to the Internet.

DYNAMIC IP ADDRESS ASSIGNMENT

Thankfully, we don't typically have to choose an IP address for every device we connect to a network. Instead, when you connect a laptop, a printer, or even a Twitter-following bubble machine, it can request an IP address from the network itself using the Dynamic Host Configuration Protocol (DHCP). When the device tries to connect, instead of checking its internal configuration for its address, it sends a message to the router asking for an address. The router assigns it an address. This is not a static IP address which belongs to the device indefinitely; rather, it is a temporary "lease" which is selected *dynamically* according to which addresses are currently available. If the router is rebooted, the lease expires, or the device is switched off, some other device may end up with that IP address.

This means that you can't simply point a DNS entry to a device using DHCP. In general, you can rely on the IP address probably being the same for a given work session, but you shouldn't hard-code the IP address anywhere that you might try to use it another time, when it might have changed.

Even the simplest computing devices such as the Arduino board, which we look at in Chapter 5, can use DHCP. Although the Arduino's Ethernet library allows you to configure a static IP address, you can also request one via DHCP. Using a static address may be fine for development (if you are the only person connected to it with that address), but for working in groups or preparing a device to be distributed to other people on arbitrary networks, you almost certainly want a dynamic IP address.

IPv6

When IP was standardised, few could have predicted how quickly the 4.3 billion addresses that IPv4 allowed for would be allocated. The expected growth of the Internet of Things can only speed up this trend. If your mobile phone, watch, MP3 player, augmented reality sunglasses, and telehealth or sports-monitoring devices are all connected to the Internet, then you personally are carrying half a dozen IP addresses already. Perhaps you have a dedicated wallet server for micropayments? A personal web server that contains your contact details and blog? One or more webcams recording your day? Perhaps rather than a single health monitoring device, you have several distributed across your person, with sensors for temperature, heart rate, insulin levels, and any number of other stimuli.

At home you would start with all your electronic devices being connected. But beyond that, you might also have sensors at every door and window for security. More sensitive sound sensors to detect the presence of mice or beetles. Other sensors to check temperature, moisture, and airflow levels for efficiency. It is hard to predict what order of number of Internet connected devices a household might have in the near future. Tens? Hundreds? Thousands?

Enter IPv6, which uses 128-bit addresses, usually displayed to users as eight groups of four hexadecimal digits—for example, `2001:0db8:85a3:0042:0000:8a2e:0370:7334`. The address space (2^{128}) is so huge that you could assign the same number of addresses as the whole of IPv4 to *every* person on the planet and barely make a dent in it.

The new standard was discussed during the 1980s and finally released in 1996. In 2013, it is still less popular than IPv4. You can find many ways to work around the lack of public IP addresses using subnets, but there is a chicken-and-egg problem with getting people to use IPv6 without ISP support and vice versa. It was originally expected that mobile phones connected to the Internet (another huge growth area) would push this technology over the tipping point. In fact, mobile networks are increasingly using IPv6 internally to route traffic. Although this infrastructure is still invisible to the end user, it does mean that there is already a lot of use below the surface which is stacked up, waiting for a tipping point.

IPv6 and Powering Devices

We can see that an explosion in the number of Internet of Things devices will almost certainly need IPv6 in the future. But we also have to consider the power consumption of all these devices. We know that we can regularly

charge and maintain a small handful of devices. At any one moment, we might have a laptop, a tablet, a phone, a camera, and a music player plugged in to charge. The constant juggling of power sockets, chargers, and cables is feasible but fiddly. The requirements for large numbers of devices, however, are very different. The devices should be low power and very reliable, while still being capable of connecting to the Internet. Perhaps to accomplish this, these devices will team together in a mesh network. This is the vision of 6LoWPAN, an IETF working group proposing solutions for “IPv6 over Low power Wireless Personal Area Networks”, using technologies such as IEEE 802.15.4. While a detailed discussion of 6LoWPAN and associated technologies is beyond the scope of this book, we do come back to many related issues, such as maximising battery life in Chapter 8 on embedded programming.

Conclusion on IPv6

Although IPv6 is, or will be, big news, we do not go into further detail in this book. In 2013, you can find more libraries, more hardware, and more people that can support IPv4, and this is what will be most helpful when you are moving from prototype to production on an Internet of Things device. Even though we are getting close to the tipping point, existing IPv4 services will be able to migrate to IPv6 networks with minimal or possibly no rewriting.

If you are working on IPv6 network infrastructure or are an early adopter of 6LoWPAN, you will have specific knowledge requirements that are beyond the current scope of this book.

MAC ADDRESSES

As well as an IP address, every network-connected device also has a MAC address, which is like the final address on a physical envelope in our analogy. It is used to differentiate different machines on the same physical network so that they can exchange packets. This relates to the lowest-level “link layer” of the TCP/IP stack. Though MAC addresses are globally unique, they don’t typically get used outside of one Ethernet network (for example, beyond your home router). So, when an IP message is routed, it hops from node to node, and when it finally reaches a node which knows where the *physical* machine is, that node passes the message to the device associated with that MAC address.

MAC stands for *Media Access Control*. It is a 48-bit number, usually written as six groups of hexadecimal digits, separated by colons—for example:

01:23:45:67:89:ab

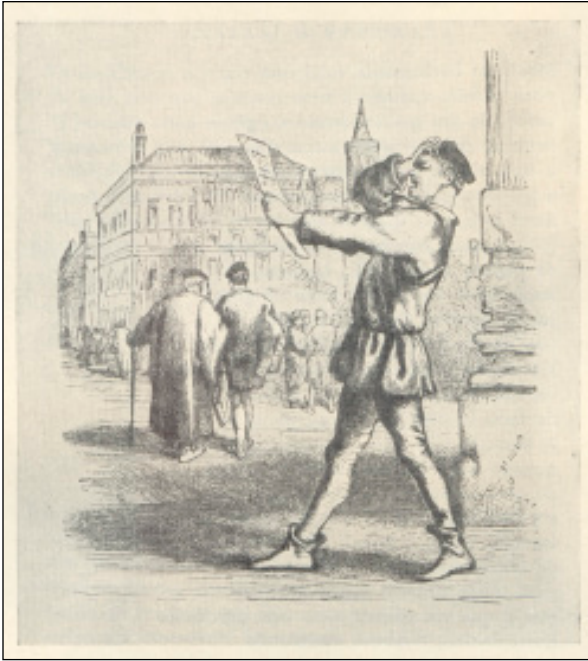
Most devices, such as your laptop, come with the MAC address burned into their Ethernet chips. Some chips, such as the Arduino Ethernet's WizNet, don't have a hard-coded MAC address, though. This is for production reasons: if the chips are mass produced, they are, of course, *identical*. So they can't, physically, contain a distinctive address. The address could be stored in the chip's firmware, but this would then require every chip to be built with custom code compiled in the firmware. Alternatively, one could provide a simple data chip which stores just the MAC address and have the WizNet chip read that. Obviously, most consumer devices use some similar process to ensure that the machine always starts up with the same unique MAC address. The Arduino board, as a low-cost prototyping platform for developers, doesn't bother with that nicety, to save time and cost. Yet it does come with a sticker with a MAC address printed on it. Although this might seem a bit odd, there is a good reason for it: that MAC address is reserved and therefore is guaranteed unique if you want to use it. For development purposes, you can simply choose a MAC address that is known not to exist in your network.

WizNet is a Korean manufacturer which specialises in networking chips for embedded devices. Many popular microcontrollers which we look at in Chapter 5 use these chips.

TCP AND UDP PORTS

A messenger with a formal invitation for a wealthy family of the Italian Renaissance would go straight to the front entrance to deliver it. A grocer delivering a crate of the first artichokes of the season would go instead to a service entrance, where the crate could be taken quickly to the kitchen without getting in the way of the masters. The following engraving, by John Gilbert, is taken from Shakespeare's *Romeo and Juliet*. This reminds us that the house of the Capulets has at least one other entrance—on Juliet's balcony. If Romeo wants to see his beloved, that is the only way to go. If he climbs up the wrong balcony, he'll either wait outside (the nurse is fast asleep and can't hear his knocks) or get chased away by the angry father.

Similarly, when you send a TCP/IP message over the Internet, you have to send it to the right port. TCP ports, unlike entrances to the Capulet house, are referred to by numbers (from 0 to 65535).



Romeo and Juliet, Act I, Scene 2, by John Gilbert, before 1873.

Public domain http://en.wikipedia.org/wiki/File:Scene_2.jpg.

AN EXAMPLE: HTTP PORTS

If your browser requests an HTTP page, it usually sends that request to port 80. The web server is “listening” to that port and therefore replies to it. If you send an HTTP message to a different port, one of several things will happen:

- Nothing is listening to that port, and the machine replies with an “RST” packet (a control sequence resetting the TCP/IP connection) to complain about this.
- Nothing is listening to that port, but the firewall lets the request simply hang instead of replying. The purpose of this (lack of) response is to discourage attackers from trying to find information about the machine by scanning every port. (Imagine Romeo knocking on the sleeping nurse’s window.)

- The client has decided that trying to send a message to that port is a bad idea and refuses to do it. Google Chrome does this for a fairly arbitrary list of “restricted ports”.
- The message arrives at a port that is expecting something other than an HTTP message. The server reads the client’s response, decides that it is garbage, and then terminates the connection (or, worse, does a nonsensical operation based on the message).

Ports 0–1023 are “well-known ports”, and only a system process or an administrator can connect to them.

Ports 1024–49151 are “registered”, so that common applications can have a usual port number. However, most services are able to bind any port number in this range.

The Internet Assigned Numbers Authority (IANA) is responsible for registering the numbers in these ranges. People can and do abuse them, especially in the range 1024–49151, but unless you know what you’re doing, you are better off using either the correct assigned port or (for an entirely custom application) a port above 49151.

You see custom port numbers if a machine has more than one web server; for example, in development you might have another server, bound to port 8080:

```
http://www.example.com:8080
```

Or if you are developing a website locally, you may be able to test it with a built-in test web server which connects to a free port. For example, Jekyll (the lightweight blog engine we are using for this book’s website) has a test server that runs on port 4000:

```
http://localhost:4000
```

The secure (encrypted) HTTPS usually runs on port 443. So these two URLs are equivalent:

```
https://www.example.com  
https://www.example.com:443
```

OTHER COMMON PORTS

Even if you will rarely need a complete catalogue of all port numbers for services, you can rapidly start to memorize port numbers for the common services that you use daily. For example, you will very likely come across the following ports regularly:

- 80 HTTP
- 8080 HTTP (for testing servers)
- 443 HTTPS
- 22 SSH (Secure Shell)
- 23 Telnet
- 25 SMTP (outbound email)
- 110 POP3 (inbound email)
- 220 IMAP (inbound email)

All of these services are in fact application layer protocols.

APPLICATION LAYER PROTOCOLS

We have seen examples of protocols at the different layers of the TCP/IP stack, from the low-level communication across wired Ethernet, the low-level IP communication, and the TCP transport layer. Now we come to the highest layer of the stack, the application layer. This is the layer you are most likely to interact with while prototyping an Internet of Things project (and we look at this in greater detail in Chapter 7). It is useful here to pause and flesh out the definition of the word “protocol”.

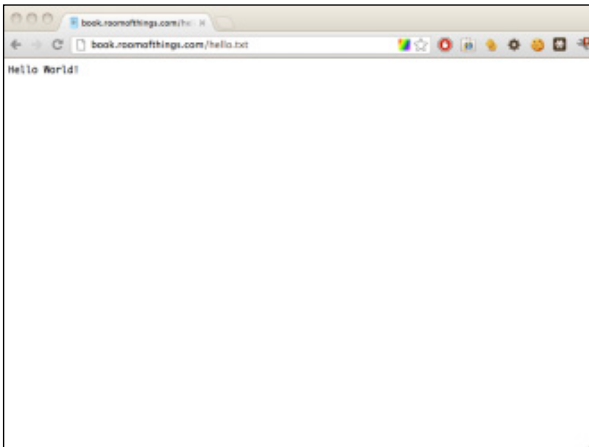
A *protocol* is a set of rules for communication between computers. It includes rules about how to initiate the conversation and what format the messages should be in. It determines what inputs are understood and what output is transmitted. It also specifies how the messages are sent and authenticated and how to handle (and maybe correct) errors caused by transmission.

Bearing this definition in mind, we are ready to look in more detail at some application layer protocols, starting with HTTP.

HTTP

The Internet is much more than just “the web”, but inevitably web services carried over HTTP hold a large part of our attention when looking at the Internet of Things.

HTTP is, at its core, a simple protocol. The client requests a resource by sending a command to a URL, with some headers. We use the current version of HTTP, 1.1, in these examples. Let’s try to get a simple document at `http://book.roomofthings.com/hello.txt`. You can see the result if you open the URL in your web browser.



A browser showing “Hello World!”

But let’s look at what the browser is actually sending to the server to do this. The basic structure of the request would look like this:

```
GET /hello.txt HTTP/1.1
Host: book.roomofthings.com
```

Notice how the message is written in plain text, in a human-readable way (this might sound obvious, but not all protocols are; the messages could be encoded into bytes in a binary protocol, for example).

We specified the `GET` method because we’re simply getting the page. We go into much more detail about the other methods in Chapter 7, “Prototyping Online Components”. We then tell the server which resource we want (`/hello.txt`) and what version of the protocol we’re using.

Then on the following lines, we write the headers, which give additional information about the request. The Host header is the only required header in HTTP 1.1. It is used to let a web server that serves multiple virtual hosts point the request to the right place.

Well-written clients, such as your web browser, pass other headers. For example, my browser sends the following request:

```
GET /hello.txt HTTP/1.1
Host: book.roomofthings.com
Accept: text/html,application/xhtml+xml,application/
      xml;q=0.9,*/*;q=0.8
Accept-Charset: UTF-8,*;q=0.5
Accept-Encoding: gzip,deflate,sdch
Accept-Language :en-US,en;q=0.8
Cache-Control: max-age=0
Connection: keep-alive
If-Modified-Since: Tue, 21 Aug 2012 21:41:47 GMT
If-None-Match: "8a25e-d-4c7cd7e3dlcc0"
User-Agent: Mozilla/5.0 (Macintosh; Intel Mac OS X 10_6_8)
      AppleWebKit/537.1
      (KHTML, like Gecko) Chrome/21.0.1180.77 Safari/537.1
```

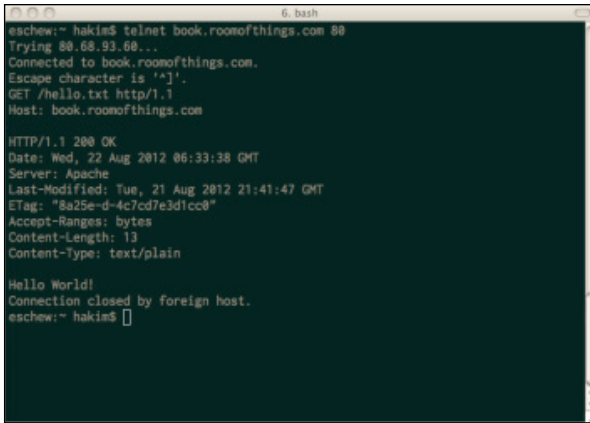
The `Accept-` headers tell the server what kind of content the client is willing to receive and are part of “Content negotiation”. For example, if I had passed

```
Accept-Language: it,en-US,en;q=0.8
```

the server might agree to give me the Italian version of the site instead, reverting to English only if it doesn’t have that page in Italian.

The other fields give the server more information about the client (for statistics and for working around known bugs) and manage caching and so on.

Finally, the server sends back its response. We already saw what that looked like in the browser, but now let’s look at what the full request/response looks like if we speak the HTTP protocol directly. (Obviously, you rarely have to do this in real life. Even if you are programming an Internet of Things device, you usually have access to code libraries that make the request, and reading of the response, easier.)



```
es Chew:~ hakim$ telnet book.roomofthings.com 80
Trying 88.68.93.68...
Connected to book.roomofthings.com.
Escape character is '^J'.
GET /hello.txt http/1.1
Host: book.roomofthings.com

HTTP/1.1 200 OK
Date: Wed, 22 Aug 2012 06:33:38 GMT
Server: Apache
Last-Modified: Tue, 21 Aug 2012 21:41:47 GMT
ETag: "8a25e-d-4c7cd7e3d1cc0"
Accept-Ranges: bytes
Content-Length: 13
Content-Type: text/plain

Hello World!
Connection closed by foreign host.
es Chew:~ hakim$
```

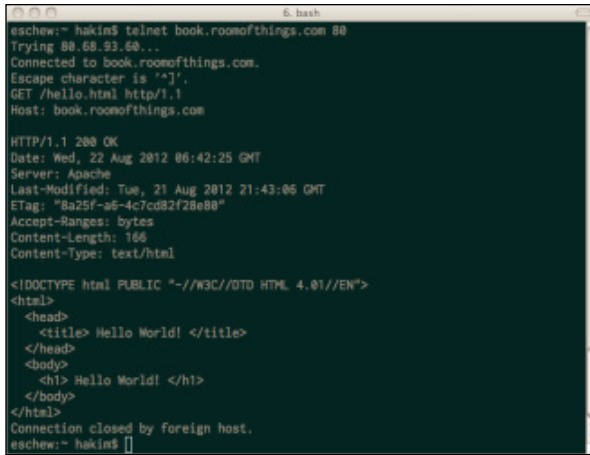
The request/response cycle.

Notice how we connect using the `telnet` command to access port 80 directly. Now that we can see the full request, it looks at first sight as if we're repeating some information: the hostname `book.roomofthings.com`. But remember that DNS will resolve the name to an IP address. All the server sees is the *request*; it doesn't know that the command that started the request was `telnet book.roomofthings.com 80`. If the DNS name `foo.example.com` also pointed at the same machine, the web server might want to be able to respond in a different way to `http://foo.example.com/hello.txt`.

The server replies, giving us a 200 status code (which it summarizes as "OK"; that is, the request was successful). It also identifies itself as an Apache server, tells us the type of content is `text/plain`, and returns information to help the client cache the content to make future access to the resource more efficient.

You may be wondering where the *Hypertext* part of the protocol is. All we've had back so far is text, so shouldn't we be talking HTML to the server? Of course, HTML documents are text documents too, and they're just as easy to request.

Notice how, for the server, replying with a text file or an HTML document is exactly the same process! The only difference is that the Content-Type is now `text/html`. It's up to the client to read that markup and display it appropriately.



```

es Chew:~ hakin$ telnet book.roomofthings.com 80
Trying 88.68.93.60...
Connected to book.roomofthings.com.
Escape character is '^['.
GET /hello.html http/1.1
Host: book.roomofthings.com

HTTP/1.1 200 OK
Date: Wed, 22 Aug 2012 06:42:25 GMT
Server: Apache
Last-Modified: Tue, 21 Aug 2012 21:43:06 GMT
ETag: "8a25f-a6-4c7cd82f28e88"
Accept-Ranges: bytes
Content-Length: 166
Content-Type: text/html

<!DOCTYPE html PUBLIC "-//W3C//DTD HTML 4.01//EN">
<html>
  <head>
    <title> Hello World! </title>
  </head>
  <body>
    <h1> Hello World! </h1>
  </body>
</html>
Connection closed by foreign host.
es Chew:~ hakin$

```

The request/response cycle with HTML.

We look at more features of HTTP over the course of this book, but everything is based around this simple request/response cycle! In Chapter 7, we look at web APIs (which are, arguably, even higher-level protocols that just happen to sit on top of HTTP) while deepening our understanding of HTTP.

HTTPS: ENCRYPTED HTTP

We have seen how the request and response are created in a simple text format. If someone eavesdropped your connection (easy to do with tools such as Wireshark if you have access to the network at either end), that person can easily read the conversation. In fact, it isn't the *format* of the protocol that is the problem: even if the conversation happened in binary, an attacker could write a tool to translate the format into something readable. Rather, the problem is that the conversation isn't encrypted.

The HTTPS protocol is actually just a mix-up of plain old HTTP over the Secure Socket Layer (SSL) protocol. An HTTPS server listens to a different port (usually 443) and on connection sets up a secure, encrypted connection with the client (using some fascinating mathematics and clever tricks such as the “Diffie–Hellman key exchange”). When that's established, both sides just speak HTTP to each other as before!

Published by Whitfield Diffie and Martin Hellman in 1976, Diffie–Hellman (D-H) key exchange is a way for two people to exchange cryptographic keys in public, without an eavesdropper being able to decode their subsequent conversation. This is done by each side performing mathematical calculations which are simple to do but not to undo. For example multiplying two prime numbers together is easily done, but if they are sufficiently large numbers, an attacker cannot factor the result back into the original primes given current computing power. Neither side ever sends their own secret key unencrypted, but only the result of multiplying it with a shared piece of information. By performing the calculation again on the key sent to them over the network, both parties end up with a “shared secret”. The Khan Academy provides a good walkthrough of this process, at Academy <https://www.khanacademy.org/math/applied-math/cryptography/modern-crypt/v/diffie-hellman-key-exchange--part-2>.

This means that a network snooper can find out only the IP address and port number of the request (because both of these are public information in the envelope of the underlying TCP message, there’s no way around that). After that, all it can see is that packets of data are being sent in a request and packets are returned for the response.

OTHER APPLICATION LAYER PROTOCOLS

All protocols work in a roughly similar way. Some cases involve more than just a two-way request and response. For example, when sending email using SMTP, you first need to do the “HELO handshake” where the client introduces itself with a cheery “hello” (SMTP commands are all four letters long, so it actually says “HELO”) and receives a response like “250 Hello example.org pleased to meet you!” In all cases, it is worth spending a little time researching the protocol on Google and Wikipedia to understand in overview how it works. You can usually find a library that abstracts the details of the communication process, and we recommend using that wherever possible. Bad implementations of network protocols will create problems for you and the servers you connect to and may result in bugs or your clients getting banned from useful services. So, it is generally better to use a well-written, well-debugged implementation that is used by many

other developers. In general, the only valid reasons for you, the programmer, to ever speak to any application layer protocol directly (that is, without using a library) are

- There is no implementation of the protocol for your platform (or the implementation is inefficient, incomplete, or broken).
- You want to try implementing it from scratch, for fun.
- You are testing, or learning, and want to make a particular request easily.

SUMMARY

The TCP/IP protocol suite is the foundation of data communication over the Internet. Each layer represents a set of rules for how to communicate, and every higher layer builds on the lower ones, offering a higher level of abstraction. Most development on the Internet, then, will involve the very highest layers of abstraction—the application layer, which includes the HTTP protocol which enables the world wide web, as well as the APIs that we explore in Chapter 7. To really understand how communication works, and to make the best technological choices, it is important to understand the lower levels, too.

TCP offers great reliability on the transport layer; however, sometimes you may decide that the lightweight UDP protocol suits your needs better.

IP addresses, an important concept from the Internet layer, are key for accessing devices across the Internet. The next generation of this protocol, IPv6, is used heavily in infrastructure and will be critical in preventing the exhaustion of addresses which is coming, hastened by the massive growth in mobile telephony, tablet computing, and the Internet of Things.

At the lowest level, the link layer, devices are identified not by IP addresses (as this isn't defined until the layer above) but rather by MAC addresses. This layer includes common types of local network, such as wired Ethernet and WiFi, as well as new personal area network (PAN) protocols, such as the 802.15.4 standard.

In the next chapter, we take a first look at prototyping a connected device, from the physical Thing itself, to its embedded electronics and the online software platform which will turn it into an enchanted object.